Universal Content Launcher

A series of tutorials detailing the creation of a content launcher in Microsoft Visual C#

Tutorial One

Building Foundations

Contents

Introduction 3
Tutorial Structure 3
Tutorial Requirements 3
Part One: A Plan of Action 4
Part Two: Starting the Project 8

Introduction

Universal Content Launcher, or UCL as we will refer to it from now on, is a project that we are going to develop to create a windows executable file that will enable you to launch content like games, applications and media files off your hard drive.

You do not need any previous programming knowledge to complete these tutorials, only a strong desire to learn. It is assumed however that you do understand basic general concepts such as what a file is and what a file path is.

Tutorial Structure

The tutorial is designed to teach the basics of the C# programming language whilst showing a practical example of a simple content launcher. Once the launcher is complete, we will start to add more complex options and features.

It would be nice to create a plan of how the tutorials are to be structured, but these tutorials will be written alongside actually coding the launcher, so we're going to be making it up as we go along – warts and all!

Tutorial Requirements

The tutorials require that you have a suitable Integrated Development Environment installed on your machine; we will refer to this as the IDE from now on. The IDE is the software package that you will use to create the UCL application, and you can choose from either Microsoft Visual Studio 2005, if you have it, or Visual C# 2005 Express, which can be downloaded free from Microsoft here: http://msdn.microsoft.com/vstudio/express/visualcsharp/download/

There are other choices for developing C# windows applications, but this tutorial will be specifically matched to the Microsoft products. The tutorials will be created using Visual Studio 2005, if you are using the Visual C# Express then there may be some subtle differences. Hopefully someone using Visual C# Express can confirm if there are any problems and the tutorial can be revised.

When you have downloaded and installed your IDE you are ready to start.

Part One: A Plan of Action

So here we are at the beginning, some of you know different languages, some of you know C# and some of you know nothing! Let me explain how we are going to go through this series of tutorials in a way that can accommodate all.

There will be no preliminary talk of variables and structures and what you need to know about C# or even programming in general. We will address things as and when they occur, if you already know about the C# programming language, then just skip over the beginner sections.

A point to note is that I myself do not know everything! There are different ways of doing things in programming, some are better than others; some don't matter depending on the problem. This is a guide to the way I would quickly create a Launcher in C#; it is not necessarily the best or most efficient way, but it is designed to be the simplest way. This tutorial will hopefully create many discussion points where methods and choices can be debated, and hopefully be the catalyst to even greater things; this is where you truly learn how to program – through the input of others.

So, let's load up our IDE and start right?

WRONG!

Even though I've written enough programs to sometimes foolishly think I can do this – it is never a good idea! First we need to give some thought to our program:

To start with, here are two simple questions:

1. What does my program need to do?
2. What does my program need to know, to do it?

Well, we know what it needs to do; it needs to create a list of files that we can then double click on to launch the appropriate programs. Sounds simple, but is it? There's more to think about:

1. How will my program show the content?
2. Does it need to scan all my drives for every available executable?
3. How does it remember them all?

There are a lot of questions that have more than one answer, for example, how do we want to show the content? In a dropdown box? In a list box with icons? In a tab control? In a tree view? In a HTML page?

These are design decisions that you as the developer make. You can choose the easiest one to implement, the one that you think will "wow" more people, or you can implement all of them and let the user of your program choose. I have always opted for the latter wherever possible, as it makes your application friendlier to the end user, but we need to start at the beginning and take it from there, so we will only initially implement a single solution. For the purpose of this tutorial, to start with, let's develop with a tree view control, as this will fit in nicely with a modern technology like XML, which you will see later.

And what about language? Do we assume everyone understands English, or do we build in a mechanism for Multi-Language support? It is far easier to code quickly in a single language, but if we do need to implement other languages in the future, then it is easier to build in from the start, particularly if you are working on a sizeable project. To keep things simple, we're going to develop in just a single language in this tutorial and demonstrate how easily this can be accomplished. As it is a relatively small project, it will not be too great a challenge to develop a Multi-Language solution after the basic launcher has been built, by using some of the methods we will learn along the way.

The question of whether we scan the hard drive for launchable program files is usually easily answered. Users hate a wait, and want control over your program, so by putting the management of the content list into the user's hands we solve both of these problems at once! At a later date we could incorporate a "new Content" scan to make their life easier – let's see how it goes, the point is to start simple and build from there.

The question of how do we store the content is a twofold one:

> How do we store it?
>
> And how do we structure it?

Structures are important things; they define how something is made up. For example the structure for "me" may say that I have a Head, body, Arms and Legs. Now if that were true I would be pretty damn ugly, with no neck and eyes and all, but with programming it's not about making things perfect if you don't need to, it's about efficiency and only dealing with the things we need to deal with. In this scenario then our structure of "me" may be perfectly adequate. In C# a structure like this is referred to as a Struct – someone got lazy!

We can take structures to a further level which we can call a class. A class is like a Struct in that it can tell me about my structure, but also tell me how to do things with that structure. For example my class for "me" may contain the same structure detailed above, plus some definitions for how I wave my arm, how I nod my head etc. These definitions are commonly known as methods.

Taking this into account (and there's a lot to take in) how can we define something for the data we need to hold for launching content off our hard drive? Well, let's have a go!

Let's think about a class for launching content, we're not bothered about the methods of the class initially (how to do things with the class), but what sort of information do we need to hold?

> The executable that we need to launch.
>
> The path where the executable is stored. It is a good idea to store the path and executable separately as we may need to do something specifically with the path, like change directory into it, before launching the executable for it to work correctly.

If the data we are storing is made up of alphanumeric characters, then it is referred to as a string (string of characters) so we would have a string for the executable and a string for its path.

Great, anything else? Well yes, sometimes we may need to pass arguments to an executable to get it to do certain things so:

An arguments string.

We also want to list it in our program. If the executable is called 1.exe then that's not going to be very informative, so let's store a name, and also an Icon that we can use when displaying it.

A name string.

An icon string – We could store this as a path and a separate icon name or just all together as one big string. As we only need to display a single icon for each of our content, then for simplicity, let's just store the whole thing.

Is that it? Let's think about more questions. What about if we launch some media? A video or mp3 for example; we need to know the path and executable of the media player to use - which we can hold with the information already defined above, but we also need to know the path and filename of the file to play. We may be able to handle this with the arguments field, but that depends entirely on the media player that is being used. This would be a bad assumption to make, so let's think ahead and store two more fields:

FilePath – path of the file to be passed to our executable.

FileName – name of the file to be passed to our executable.

This way our folder and executable fields will always store the content being launched from UCL, and the FileName and FilePath will store the details of any file that needs passing to it. Sometimes it's good to think forward like this – maybe we will end up not using it, but it's easier to take out than to put in!

And what about Steam Content? Surely we need to store a load of other stuff for that? Well, you're right! But don't worry too much about this; there are some neat ways of extending our original Content Class to accommodate Steam. This is known as inheritance and will be covered when we develop our Steam solution. But it would be prudent to store a unique type against our content so that we can sort content and even do specific things with it depending on its type.

Ok, so let's give our content data some meaningful names:

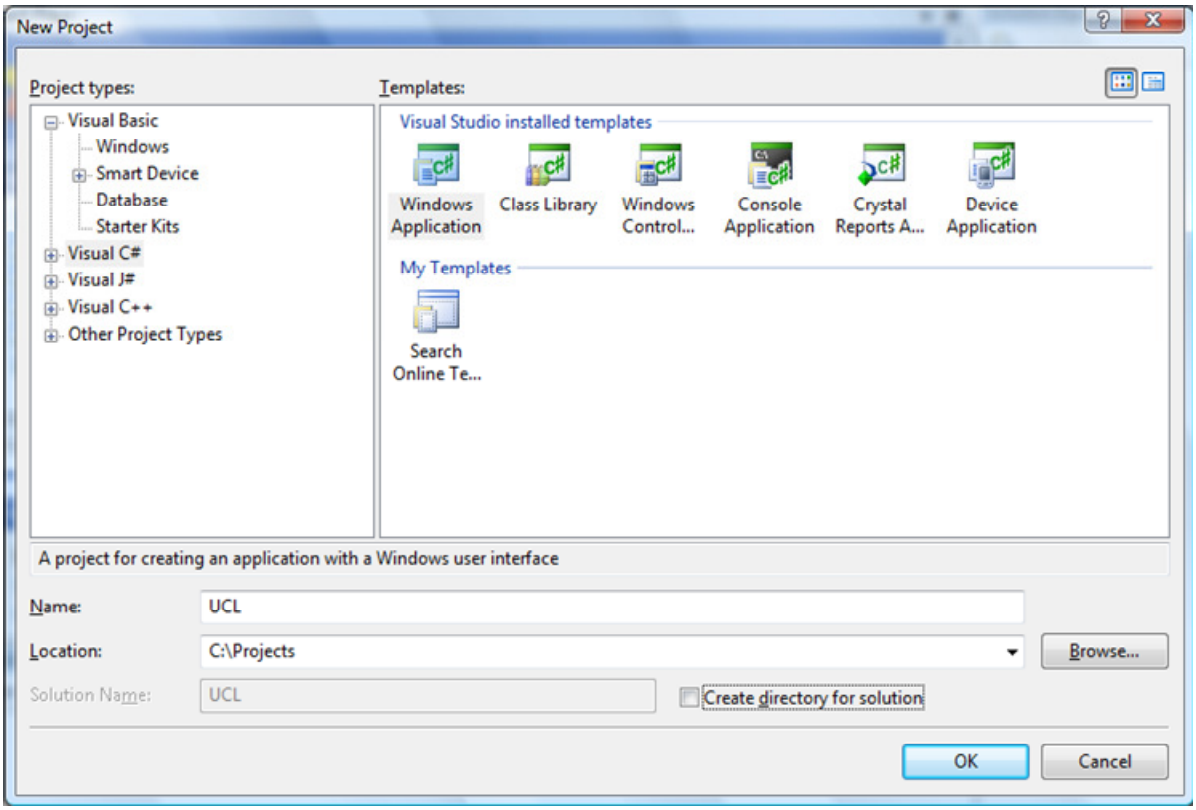
- Type
- Name
- Icon
- ExeName
- ExePath
- FileName
- FilePath
- Arguments

Of course there are probably a lot more things we can do, but it's a start and we can easily change it later without too much trouble, and probably will! The point of these tutorials is to build UCL as we go along, that way if we need to change anything, then we'll see how to do it as and when we need to.

Phew! Now we've got that sorted, let's start to code UCL!

Part Two: Starting the Project

Ok, we want to start a new project, so start up your IDE and create a new project (In Visual Studio 2005 you can do this on the left hand side of the home page, by clicking on the link called “Create Project...”)



You are now creating a new project. You may not have all the Project types listed above, but as long as you have the Visual C# entry, then you can continue:

Click on Visual C# entry in the left hand Project types panel, and this will show you all the possible types of C# projects we can create in the Templates window on the right.

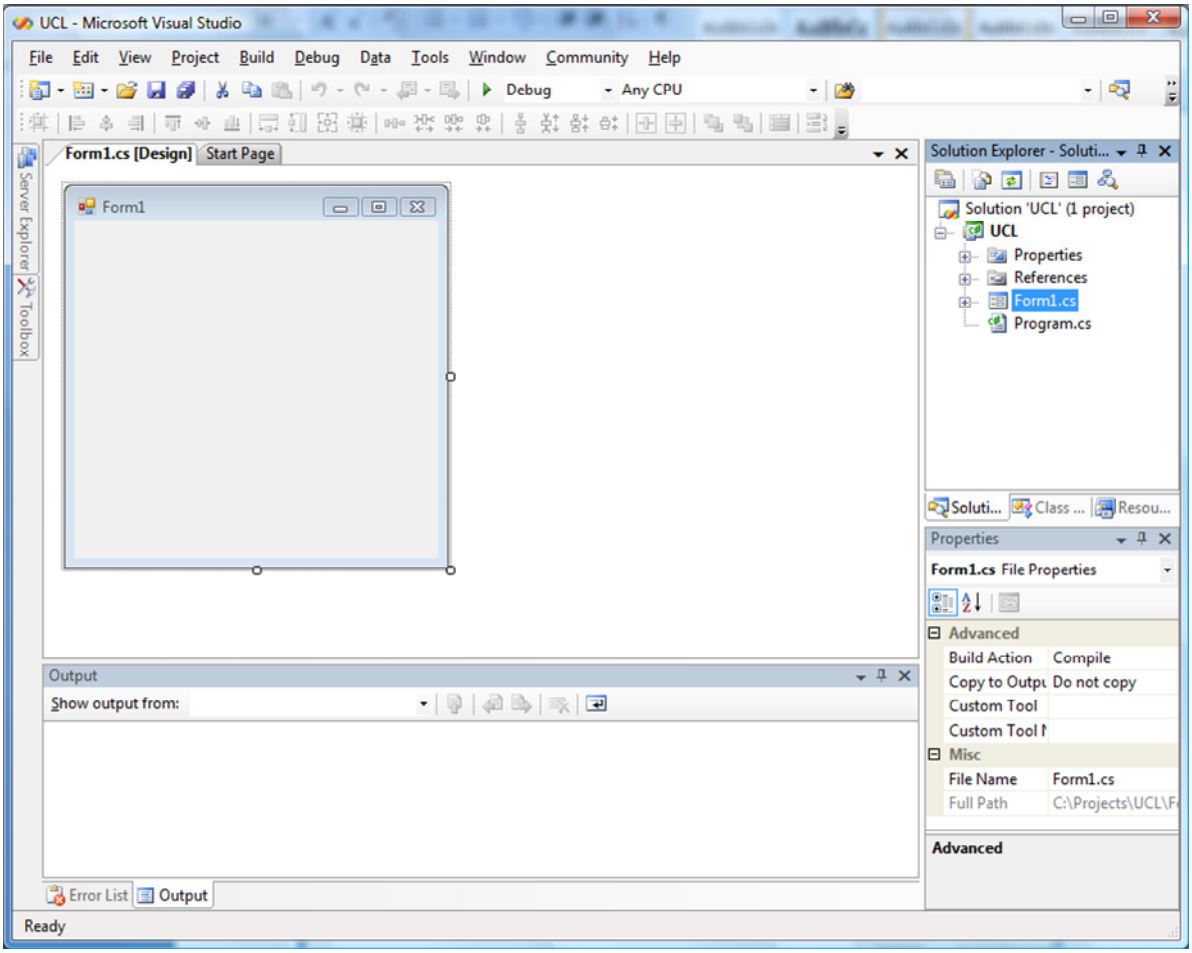
Click on Windows Application to select it.

Give your project a name at the bottom of the screen. I have chosen UCL as the project name, now you could have typed anything in here, like “Universal Content Launcher”, but you need to be wary that you may need to type this in quite a few times in your code. I’m lazy, so there you go!

Select the location where you want to save your project to. If you chose to call your project UCL and select the location C:\Projects then your project will be created in C:\Projects\ UCL\

In Visual Studio 2005, you can also choose to save the solution files to a folder that will then contain the Project folder. The solution files are just a set of files that Visual Studio uses to package everything together. I usually untick “Create Directory for Solution” as I like it to sit in the root of my project folder. It is personal preference however, and if you selected to create a directory for the solution and gave it the name of “Universal Content Launcher”, then your project files would end up being structured like so: C:\Projects\Universal Content Launcher\UCL\

Click OK and your Project will be created and you will be presented with a blank form called Form1.cs as shown below. (No prize for guessing what cs stands for!)



Let’s look at some of these windows we have in front of us.

To the left you can see two tabs: Server Explorer and Toolbox. For this project we won’t be using Server Explorer, but we will be visiting toolbox quite a bit as this is where we can select controls to put on our form (things like labels, buttons, text boxes etc)

Next to that we can see the form we are currently designing in the main windows in a tab called “Form1.cs[Design]”, there will also be other tabs in this section as we progress, that will allow us to quickly move through the different files in our project.

Below this are the output windows – we will discuss these at a later date.

On the right we can see the solution explorer. This shows you all the files that are being used within your project, so yes, you really did create all them when you started the new project!. As you add new forms and code, then they will appear here. Notice there are tabs under the solution explorer that allow you to quickly move between the solution, class and resource views. We'll talk more about these later.

Below this is the Property Window, where you can set various properties to different values for the currently selected object. At the moment we have Form1.cs selected, Form1 isn't a very informative name, so let's change it to "UCLMain" by clicking in the "File Name" box and changing "Form1.cs" to "UCLMain.cs" then pressing return. If "File Name" is not in the property list then click on Form1.cs in the Solution Explorer and try again. Notice that the name changes not only in the property window, but also in the tab above the form in the main window and in the solution explorer. It is important to change form names in this way, there is a lot of hidden code in your project which references the form by name, that will be updated correctly using this method.

Hang On! Why does the form itself still say Form1 in its own title bar? Well that's because we changed the name of the file that holds all the details about Form1, but we haven't altered any of the properties for the form itself. (I.e. the things we will see when our application runs)

Click anywhere on the actual form in the main window, and you will notice that the listed properties in the property window have changed. This is because different objects can have different properties associated with them. Scroll down through the properties until you see the Text Property. Change the text from "Form1" to "Universal Content Launcher" and hit return.

Are you happy now?

Notice how the output window at the bottom has gradually been filling up with information whilst we were working? This is just our IDE telling us what we have done, there are other windows that will be displayed here in the future that are of more use to us, and we will discuss them at a later date.

You may also see that UCLMain.cs has expanded in the solution explorer to show UCLMain.Designer.cs and possibly UCLMain.resx. I recommend you largely ignore these files until you know what they are for – we can talk about them later.

Ok, so we see from the solution explorer that there is now:

- A Properties folder (let's leave it for now)
- A References folder (again, leave it!)
- Our UCLMain form.
- And a file called Program.cs. This is a file created by the IDE to enable our program to start correctly, double click on it to open it up, and let's have a look at the code.

What is all that! Well, don't worry, we can explain it all as we go along at times when it will make more sense, however this line is the one we are currently interested in:

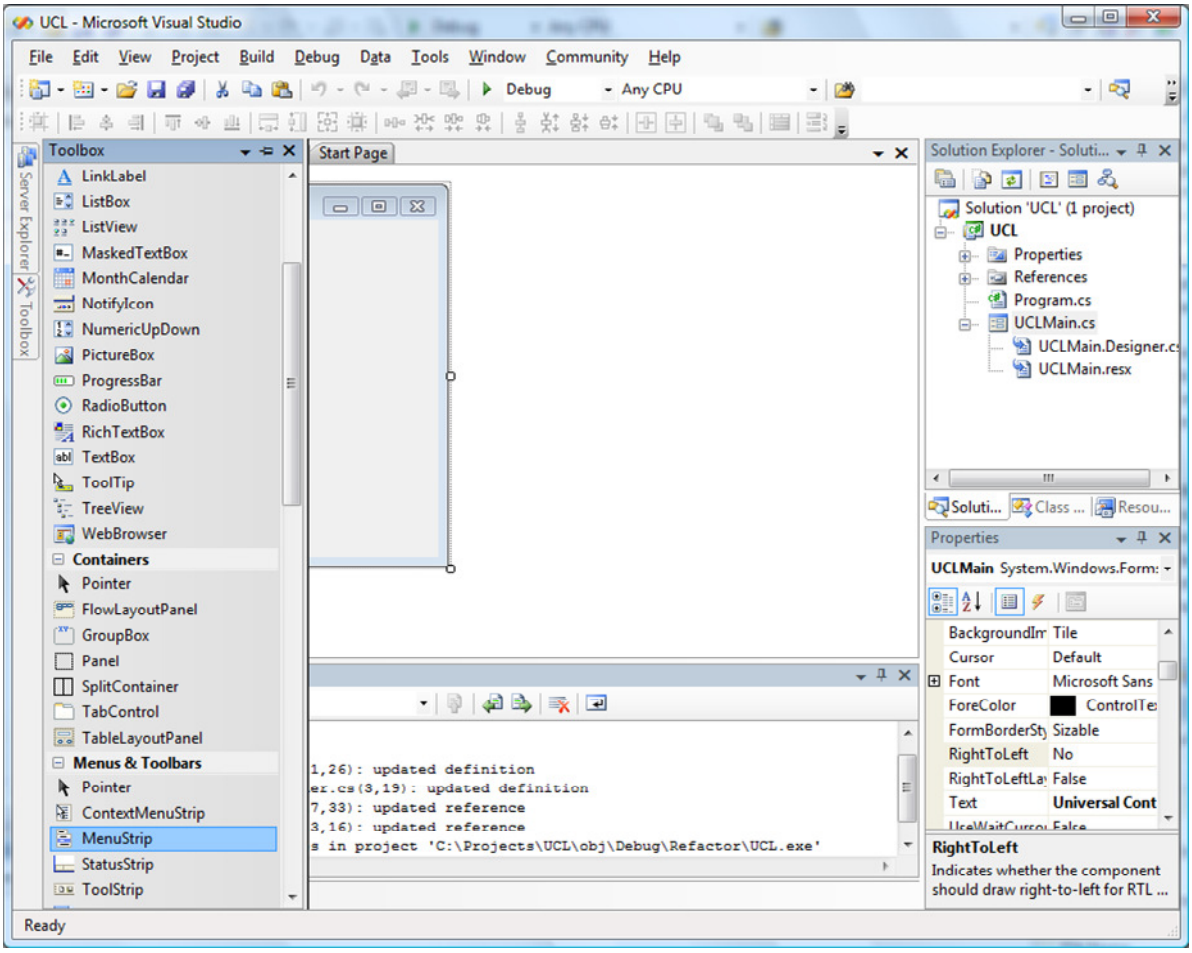
```
Application.Run(new UCLMain());
```

This tells our application (UCL) to run and display a new instance of the UCLMain form, so if we run our project we should see our form. Let's do it!

At the top of the IDE window in one of the toolbars, there is a green triangular "play" arrow next to the word "Debug" – press it and lets run our program. A lot of stuff flashes by, but eventually you get to see UCL running in its own window (with the correct name I hope!) you can close UCL either by clicking on its close button, or by clicking on the blue "stop" square that is located in the toolbar where we originally clicked the green "play" arrow.

Debugging? But isn't that where you can see the program running and look at what it's doing and fix things up? Well, Yeah – that's right! But because we haven't written anything to go wrong yet, then it all went smoothly and our project ran correctly! We'll talk about debugging a lot later on. For now, all you need to know is that to see your project working, hit the green triangle! Whenever I say "Run UCL" throughout the tutorial then this is what I want you to do.

Next let's add a very basic menu. First make sure you've selected the UCLMain form in the main window again so you are looking at the forms design. A menu in C# is just a type of control like a button, a label or a text box. All the different types of controls are accessible through the toolbox, so click on the Toolbox and scroll through it until you find the MenuStrip control.



Note that there is also a control called ContextMenuStrip – if you don't know, then this is a menu for when you right click on something rather than a menu that appears at the top of a form.

Ok, so we've clicked on the MenuStrip control, now simply drag it onto the UCLMain form design. It will position itself at the top of the form (where you would expect a Menu to be) with a box inside it that says "Type Here".

Who are we to pass up an offer? So type into the box: &File

Note that as soon as you click in the box, two more boxes appear, one to the right and one below. This is how you create the menu, you simply just keep typing into the boxes to create the menu items, and even sub menu items that you want.

So, why did we type in "&File" and not just "File"? Well, we could have done either, but the & tells our project that we want a hotkey defined for this menu item and the character immediately after the & becomes the hotkey identifier. A hotkey is the thing in windows where you can press Alt+something to do something specific, and in the menu you will see an underscore under the hotkey identifier, so in this case you will see the menu item File, and pressing Alt+F will select the file menu in UCL when it is running. You can try this by running UCL, but first we need to put something else in the menu so you can actually see that it does something!

In the box below the &File box, type in: E&xit

If the box is no longer there then just click in the &File box, and the box beneath it will become visible again.

Do you see what we've done here? We've allocated the hotkey Alt-X to exit our program so if the user types in Alt-F then Alt-X our program will exit by selecting the File menu and then the Exit menu item. Do we need to include these hotkeys in our project? Well you will be surprised how many people have become used to them and will complain if your application doesn't support hotkeys! It's always useful to know.

Now we can see or menu by running UCL, go ahead and try it!

Oh dear! – Yeah that's right, the menus respond to the hotkeys, but UCL doesn't close when we press Alt-X!

Why? Because we haven't told UCL what to do when either Exit is clicked on the Menu or Alt-X is pressed, so it's doing nothing!

To correct this we need to write some code to tell UCL when we want it to exit. We can simply double click on the menu item that we want to add code to, so double click in the &Exit box and a new window will open with the following piece of code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace UCL
{
    public partial class UCLMain : Form
    {
        public UCLMain()
        {
            InitializeComponent();
        }

        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {

        }
    }
}
```

Now, that's a lot of code for just clicking a menu item and we haven't even put in our bit yet! Well that's right, but what has actually happened is that our IDE has created a new code method called `exitToolStripMenuItem_Click` to run our code whenever "Exit" is clicked on the File Menu. Because the IDE created this for us, it assumed we wanted to see it, and displayed the UCLMain code file for us type our code into. So we now have two UCLMain.cs files in the top row of tabs; "UCLMain.cs" and "UCLMain.cs [Design]", you can always select either the code or design file for a form by right clicking on the form in the solution explorer and choosing either "View Code" or "View Designer". Since the IDE automatically creates these methods for you if they don't already exist, then we can easily drag and drop controls onto our form designer and then simply double click on them to add the code that defines what should happen when they are used in the running program. Easy huh?

So the only bit we are interested in at the moment out of all the code above is this:

```
        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {

        }
```

And indeed that is where our cursor will be flashing (between the two braces {}) waiting for us to start typing in our code. But let's explain a bit about what all the code above is about, firstly what are all those `using` statements? The `using` statement is a bit strange in C# as it can be used in two different ways. Luckily the use of it here is the easier of the two, as it merely tells C# to include the named "Namespace" in our code – but what does that mean?

Well, just like we can write our own classes and methods to make our program simpler and easier to code, so can Microsoft (or anyone else for that reason) and they have kindly written a lot of classes within the .NET framework that allow us to do specific things so that we don't have to write them ourselves. There are classes that allow us to read and write files, classes that enable accessing network resources and so on, and all we need to do to have access to these lovely things is include them in our code.

So for example, to check if a file exists in C# we can do this:

```
if (System.IO.File.Exists("c:\\test.txt"))
```

This line will tell C# to check if the file "test.txt" exists in the root of out C drive ("C:\") In C# you have to write file paths with a double backslash for them to be valid, C# will interpret the above line as "C:\test.txt". Now, if we include the System.IO Namespace with "`using System.IO`" at the beginning of our code, then we can do this:

```
if (File.Exists("c:\\test.txt"))
```

We've told C# at the beginning of our code that anything we type in may be found in the `System.IO` Namespace, and therefore it knows exactly where to look if we abbreviate the `File.Exists` statement. The net result is that the code is far easier to read, making the statement seem more understandable, and also quicker to type! `using` statements should always appear at the very top of the code before anything else, so C# knows where to look, to find the methods you are using.

We don't need to go into detail here at the moment, but as our IDE included all these `using` statements, then it's probably a good idea to leave them there for now – And indeed if you were to remove them, then you would be faced with quite a few errors! A good reference for all the available classes that Microsoft support in the .NET framework can be found here: http://msdn2.microsoft.com/en-us/library/aa388745.aspx

After the `using` statement, we have the `namespace` declaration that shows that all the code we write in this code module (UCLMain.cs) will be wrapped up in the UCL namespace. This will be discussed in more detail throughout the project and namespaces will gradually reveal their purpose more clearly to you.

Now then, there are a lot of braces in the code above and there is a reason for this. In C# whenever we do something like declare a class or start a block of code it is enclosed in braces {} so that C# knows where the beginning and end of things are. Note that braces are automatically indented by the IDE, which makes your code easier to read.

So, in the above example we have two methods (`UCLMain()` and `exitToolStripMenuItem_Click()`) that are contained in a class called UCLMain that is itself contained in a namespace called UCL.

If this is confusing, then don't panic - it will all become clearer when we start to write some more involved code, for now let's just tell our application to close when the "Exit" menu Item is pressed. Make sure you click between the braces for the exitToolStripMenuItem_Click method. Then type in:

```
Application.Exit();
```

There is an important thing to note here, and that is that the C# programming language is case sensitive. This is true for statements and even the names you give to your own objects. So, a class called "UCLContent" is a completely different object from a class called "UCLcontent". If you get errors when trying to run your code, then always check the case of the offending line. Note also that the IDE will colour known keywords or classes differently, hence why the word "`Application`" is coloured cyan and "Exit()" isn't.

The other thing to note is the dot or full stop in the above statement. This indicates that "Exit()" is a property or method of the object "`Application`". We can differentiate between properties and methods by the fact that a method can be identified as having brackets after it. Think of a property as something that belongs to an object and holds a value that we can retrieve or sometimes even set. Think of a method as something that belongs to an object that can execute a certain action, so the method "Exit()" causes "`Application`" to exit, hence closing our application.

So how do you know what all the possible properties and methods of a given object are? Well, most modern IDE's will tell you on the screen as you're typing. So when you type in "`Application`", as soon as you type in the ".", then a list will popup of all the available methods and properties that can be used with that object. This kind of information will also be available for any classes that you write that have their own properties and methods.

Your code should now look like this for the `exitToolStripMenuItem_Click` method:

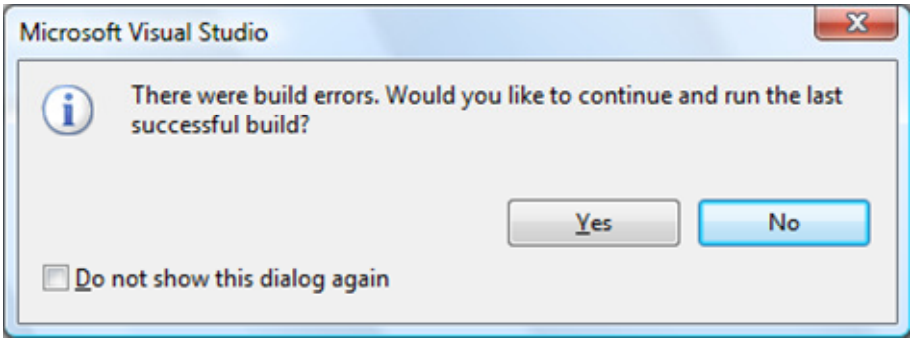
```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

Notice that we told our application to exit in a very easy way and then added a semi-colon after the statement. The semi-colon tells C# that we have entered everything we wanted to for that piece of code and to process it and go on to the next line. Most things in C# will need a ";" after them or you will get what is known as a compiler error.

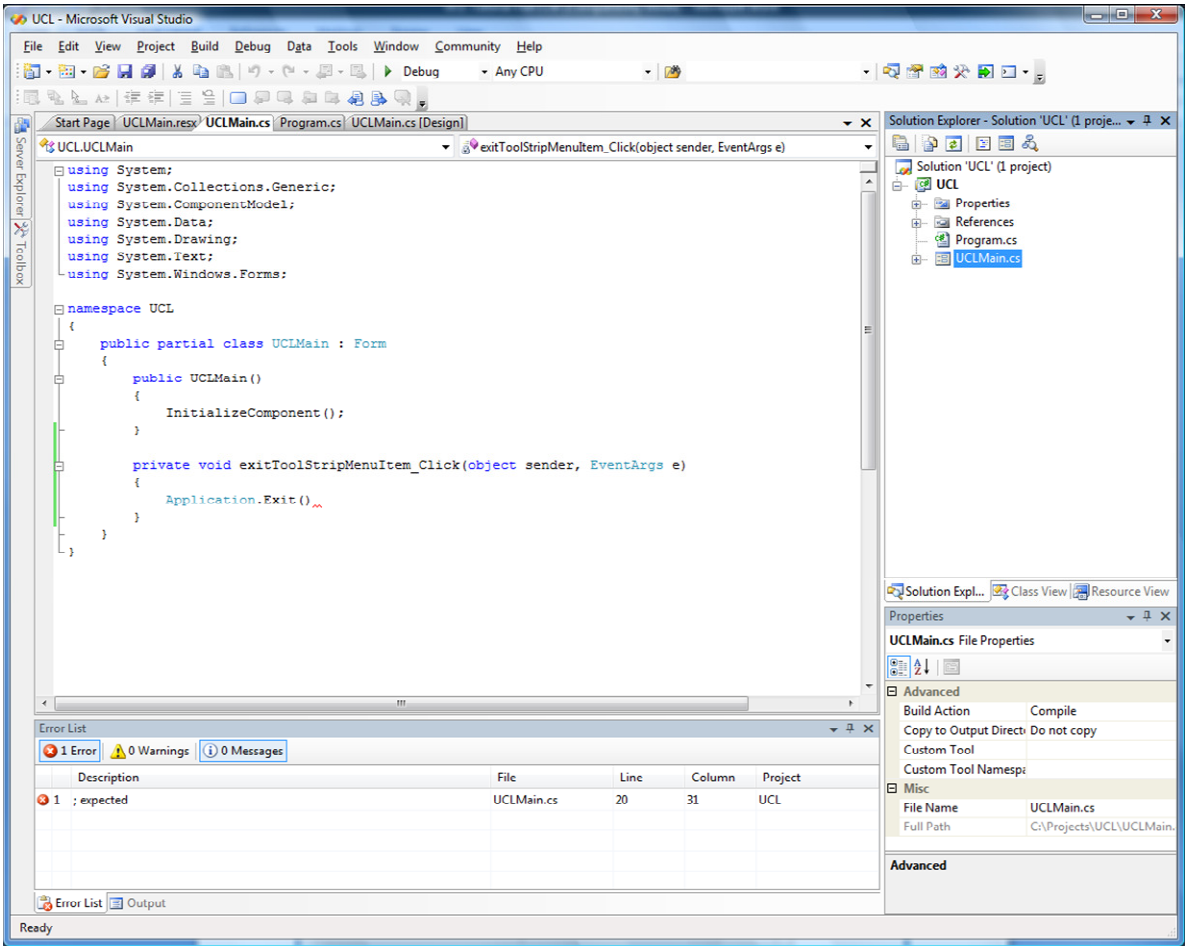
Let's see what a compiler error looks like, by removing the ";" from the line above so it looks like this:

```
Application.Exit()
```

Now try to run UCL. You should see something like this:



Click on "No", as it is pointless to run the last build that worked – we want our new code to work, so we need to find out where the problem is. Now you should see a screen similar to the one below:



Notice that our IDE is now showing an error at the bottom of the screen in the output window called “Error List”, and for this error it’s pretty self explanatory what the problem is. You can usually double click on an error to go to the line of code that is at fault, but in this case it is already in view in the main window, and you will notice that the IDE has also placed a red marker where the problem is - after Application.Exit().

That was easy, huh? So let’s put back the “;” at the end of the line and run UCL.

Now press Alt-F and then Alt-X.

Did it work? Congratulations, you’re on your way!

Let’s take a break for now, so far we’ve learned how to create a new C# project, how to create a windows menu and attach code to the menu items, and how to run and test our program. The next tutorial will start to get more involved, and we will begin to define the data structures that our program will need, and a file format for storing them.